2017

Evolving Art: Modifying Context Free Art with a Genetic Algorithm

Marina Kent
Scripps College

Pomona College
Department of Computer Science

Evolving Art: Modifying Context Free Art with a Genetic Algorithm

Marina Kent

April 21, 2017

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Peter Mawhorter, advisor

Abstract

Context Free Design Grammar (CFDG) is a programming language for defining recursive structures that can be used to create art. I use CFDG as a design space for genetic programming, experimenting with various options for crossover, mutation, and fitness. In this exploratory work, multiple generations are manually assessed to determine the usefulness of the mutation strategies and fitness functions. I find that simple value mutation and fitness that alters general program structure is not enough to produce an increase of interesting images in CFDG. I discuss these findings as well as future avenues of inquiry for genetic programming in artistic domains.

Acknowledgments

I would like to thank Professor Nancy Macko for her input and revisions of this manuscript. I would also like to thank Professor Peter Mawhorter for his guidance, support, and endless patience in helping me with python syntax.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I experimented with Context Free Art, a program for creating art using Context Free Design Grammar, as a design space for testing a genetic algorithm. Genetic algorithms (GAs) are an approach to optimizing programs, paralleling natural selection in nature. The optimization occurs because of this "natural selection:" programs that are better (have a higher fitness) are more likely to move on to the next generation. For more information on genetic algorithms and Context Free Art, see Sections 2.1 and 2.2, respectively.

The purpose of this research was to investigate whether I could create and improve upon new images with the algorithm that were interesting or aesthetically pleasing. I found that the most aesthetically pleasing results were created after just one generation, and the difficulty of accounting for what makes a program interesting resulted in fitness functions that did not optimize as expected. Because of this, further research can be done in experimenting with fitness functions in order to improve optimization of "interestingness."

In *A Step Towards the Evolution of Visual Languages* and *Graph-Based Evolution of Visual Languages*, CFDG is investigated as a design space for evolutionary algorithms [MN10] [MNR10]. In the former, hand-coded images are run through a presented "Evolutionary Art engine," to assess the power and potential of their proposed system [MN10]. In the latter, randomly created grammars are used instead of hand-coded ones, to see if a less ideal starting point would result in optimized images. Again, this study examines the adequacy of their mutation and crossover functions [MNR10]. Both of these studies strongly parallel the work done in this project; all are looking for ideal crossover and mutation functions in order to create the best images. In this project, I wanted to try to replicate these studies and further explore trade offs in crossover, mutation, and fitness.

This paper is laid out as follows: Chapter 2 describes genetic algorithms, their history, and a simple example application. It also introduces Context Free Art, a program for creating art with Context Free Design Grammar. Chapter 3 describes the layout and logic behind my genetic algorithm. Chapter 4 evaluates the effectiveness of the algorithm, and Chapter 5 summarizes the results.

Chapter 2

Background

2.1 Genetic Algorithms

Genetic algorithms are a subset of evolutionary algorithms, which were produced by researchers trying to solve problems by imitating nature. While neural networks imitate individual brains, genetic algorithms imitate genetic mechanisms within a population. This project uses genetic programming, a technique where computer programs are treated as genetic code, and a genetic algorithm is used to alter them.

Evolutionary algorithms are a type of probabilistic search and optimization algorithm modeled after organic evolution. Some evolutionary algorithms include *genetic algorithms*, *evolution strategies*, and *evolutionary programming*, all of which were developed separately but have similar fundamental ideas. Rechenberg developed evolution strategies in the 1960s and 1970s; this idea initially involved two "individuals:" one parent and one offspring, which was a mutation of the parent [Mit98]. This concept later incorporated multiple individuals, as well as genetic recombination. Evolutionary programming was developed by Fogel, Owens, and Walsh in 1966; similar to early evolution strategies, evolutionary programming only involved mutating the programs [Bac96] [Mit98]. The genetic algorithm was invented by Holland in the 1960s and he developed the idea over the 1960s and 1970s along with his colleagues and students at University of Michigan. Unlike the other two evolutionary algorithms, Holland created genetic algorithms to model evolution in nature, it was only afterwards that it was used as an optimization tool for engineering problems [Mit98].

A genetic algorithm takes a set of individual solutions to a given problem (modeling a population), each with an associated fitness value, and transforms them into a new population (modeling the next generation) using operations modeled after reproduction and natural selection, along with mutation and genetic recombination. For readers without a background in biology, fitness is a term used to describe the reproductive success an organism has in passing its genes on to the next generation; a population is a group of such individual organisms; mutation is when the genetic structure (DNA) is changed; and

genetic recombination is an exchange of DNA sequences that occurs during the formation of a cell in sexual reproduction. Genetic algorithms often involve large populations, such as hundreds or thousands of computer programs, which are bred, using genetic recombination and mutation, to create the next generation. The fitness of each generation should continue to increase, eventually reaching a point in which the algorithm produces a more desirable optimized program. The genetic algorithm models natural evolution, involving a population of individuals, each with an associated fitness, that will produce future generations that are different due to recombination and mutation. The algorithm will choose which programs to breed based off of this fitness, paralleling natural selection.

2.1.1 Simple Genetic Algorithm Example

A simple GA will start with a randomly generated population of n organisms, each with their own chromosomes, which represent the candidate solutions to a problem. The candidate solutions can be randomly created or manually chosen, depending on the problem. For example, if the GA is meant to optimize a problem for which there are already 10 possible solutions, those would likely be the candidate solutions. For cases in which candidate solutions do not already exist, they might be randomized instead. These individuals will each have a fitness associated with them, as specified by the programmer. Offspring will then be generated by selecting a pair of parents, with higher probability based off of higher fitness. Parents can be chosen more than once (done "with replacement") [Mit98]. Crossover will occur with probability P_c at a randomly chosen point (this "point" will be a random bit in the chromosome), and two offspring will be produced. Figure 2.1 depicts this process [Whi94]. Common crossover techniques include *single point*, *two-point*, and *uniform* [SP94]. In single point crossover, a point in the encoding is selected, and the two lines of code are swapped; for example, if crossover occurred at the zeroth index, and the two encodings were *11111* and *00000*, the resulting encodings would be *01111* and *10000*. Two-point crossover is similar, but instead a section of code between two points is swapped; for example, with the same encodings, if the two points were at the zeroth and third indices, the result would be *10011* and *01100*. In uniform crossover, each bit has a probability of being swapped with that of the other encoding; this could produce random encodings such as *10110* and *01001* (note that the two children, combined, still contain all of the parent's genetic information). After crossover, the two offspring will be mutated with probability P_m at each locus (bit). Mutation, in binary encoding, will be accomplished by bit inversion; for example, if *111* mutated on the zeroth and first index, it would become *001*. This reproduction process will be repeated until n offspring have been produced, completely replacing the parent generation [Mit98].

For example, an initial, randomly generated, population could look like the encodings in Table 2.1. This population consists of binary bits, which is common for GAs, but genetic algorithms are not limited to operating on binary values. In this example, the fitness it measured by the number of ones in the chromosome [Mit98].

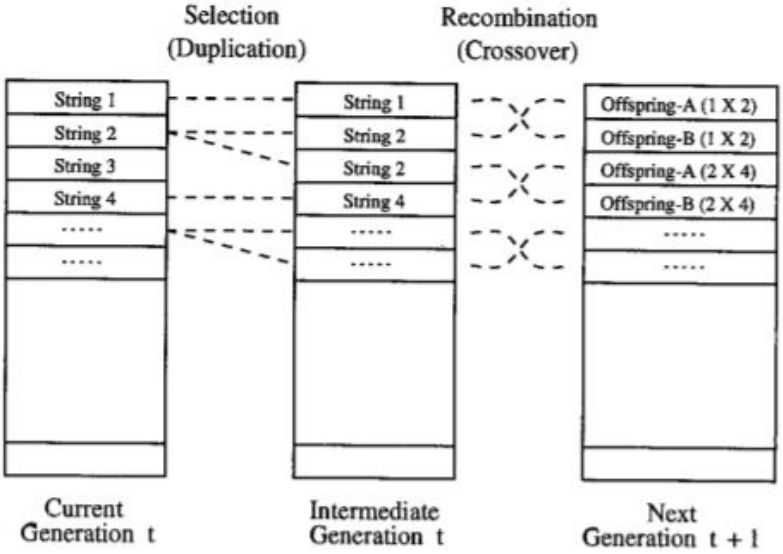

Figure 2.1: Strings being selected and "reproducing," with crossover

Chromosome label	**Chromosome string**	**Fitness**
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

Table 2.1: **Initial randomly generated population**

A common GA selection method is fitness-proportionate selection, where an individual with a higher fitness would have a higher chance of reproducing. This chance would be its fitness divided by the average fitness of the population. A common implementation of this is roulette-wheel sampling [Gol89]. Imagine a pie chart, with each individual being represented by a portion equal to its likelihood of reproduction.

For example, with this population, this roulette wheel would be spun n times, and the first two spins chose B and D to be parents, and the second two chose B and C to be parents, they would preform a single point crossover, with probability P_c, to form two offspring. Crossing over does not always result in a new offspring. For example, if B and D crossover at the first bit to form E = 10110110 and F = 01101110, and B and C do not crossover, then C and B would remain. Mutation then occurs, with probability P_m, in this example E mutates at the sixth bit, and B mutates at the first. The new population is depicted in Table 2.2 [Mit98].

Chromosome label	Chromosome string	Fitness
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Table 2.2: **New population (after potential crossover and mutation)**

Although the best string, the one with fitness 6, was lost, the average fitness rose from 12/4 to 14/4. Eventually this procedure would result in a chromosome string made entirely of ones. This string of ones in itself is not especially useful, but illustrates the concept of an overall increased fitness.

Genetic algorithms do not always return the best solution, they simply return the solution with the current highest fitness. For example, GAs can be used to "search for solutions" by taking a randomly generated solution, and systematically testing mutating each bit until one of the mutations results in a higher fitness [Mit98]. If no better fitness is found, a new solution is generated and the process restarts. Otherwise, the solution with the increased fitness is then put through the process again, hopefully increasing its fitness again. This is repeated a number of times, depending on how many generations the programmer specifies, creating new randomly generated strings, until the program is done, where it will select the most fit solution to return.

2.1.2 Art

It is easy to imagine how genetic algorithms can be used to produce art, given the right language. This "design space," the combination of changeable input parameters in the code, is the genotype, and the execution of the design space – creating the visuals – would be the phenotype. In their book, *Modeling creativity and knowledge-based creative design* [GM13], Gero and Maher address that "a certain amount of 'playful' experimentation leads to the recognition of interesting results," but assert that randomness alone is either not enough or too much experimentation. Random mutation in this design space "has the potential to produce structures that might have been impossible to imagine but [...] it is much more likely to produce meaningless products" [GM13]. Because of this randomness, GAs involving design should strive for more control and less randomness: a "conscious goal-directed process" [GM13]. My project uses the method proposed by Gero and Maher to evolve the design space, and encounters many of the difficulties addressed about randomness.

2.2 Context Free Art

This project uses Context Free Art (CFA), a digital art program, to produce images to use in a genetic algorithm. CFA takes a description of an image, written in Context Free

Design Grammar (CFDG), a language created by Chris Coyne, which consists of a root shape (called a `startshape`), and applying rules to the shape to add complexity to the image [CC13]. These rules tell Context Free how to draw a shape in terms of other shapes, often recursively. The genetic algorithm is able to breed CFDGs by accessing these shapes and rules, and treating the code of the CFDG as DNA.

CFDG has been used for research in Evolutionary Art Systems (discussed in Sections 1 and 2.1.2), and other research involving Context Free Grammars. An example of such research is that done by Christensen in *Structural Synthesis using a Context Free Design Grammar Approach* [Chr09]. He examines CFDG as a set of formal grammars modeling two-dimensional structures using a simple set of primitives, and extends these ideas to work in three dimensions. Similarities and differences in CFDG and other formal grammars are discussed, such as how Context Free Design Grammar extends the principles of syntax of the formal grammars by including transformation operators, but parallels other formal grammars by allowing for many possible substitution options in each non-terminal symbol. His extension of CFDG, Structure Synth, derives syntax from CFDG, but changes the termination criteria, adds transformations, additional colors, and a "rule retirement system," removes the `startshape` declaration, and changes the syntax relating to curly and square brackets [Chr09].

Chapter 3

Program Description

I created this program to evolve CFDG code using various crossover, mutation, and fitness functions. The input of this program is translated code taken from four images on contextfreeart.org [zee] [cra] [mom] [Chr], and its output is made of a combination of code taken from the two input parents. This outputted code can then be translated into CFDG code to make into images, which I later assess in order to determine the usefulness of my functions (see Section 4).

3.1 Program Structure

Figure 3.1 provides a visual representation of the components of a `Program` (all of the code makes up a single CFDG program). Objects are modeled after the structures used in CFA. At the highest level, paralleling the `startshape`, is a `Program`. This is what keeps track of the `startshape` name, along with all of the shape rules.

In CFA, a program can call a rule, and this rule can occur multiple times in the program with different weights, so there is a different chance of calling each version of the rule each time it is needed. Rules in context free are named shapes that the user defines, to allow for mutual recursion. The way this is accounted for in my work is by creating a `NonTerminal`: an object that keeps track of its parent, a `Program`, and its children, `ShapeDefs`.

`ShapeDefs` must know about their parent (a `NonTerminal`), and about their content (calls to rules or primitive shapes). Both primitive shapes and rules inherit from `Shape`, which inherits from `Node`. These levels of abstraction help with copying, as well as adding extra organizational structure for manipulating the objects in the genetic algorithm. Primitive shapes (`Square`, `Triangle`, or `Circle`) inherit from `SimpleShape`; they all take a number of arguments that modify their appearance. These arguments are all defined as `Modifers`. Not every CFA modifier is added to this project, but enough are that fairly complex programs can be easily translated. `Modifiers` either take one, two, three, four, or six values following their name. For example, `SQUARE [r 30]` will produce a square rotated 30 degrees, and `TRIANGLE [s 0.4 10]` will produce a triangle that is 0.4 units

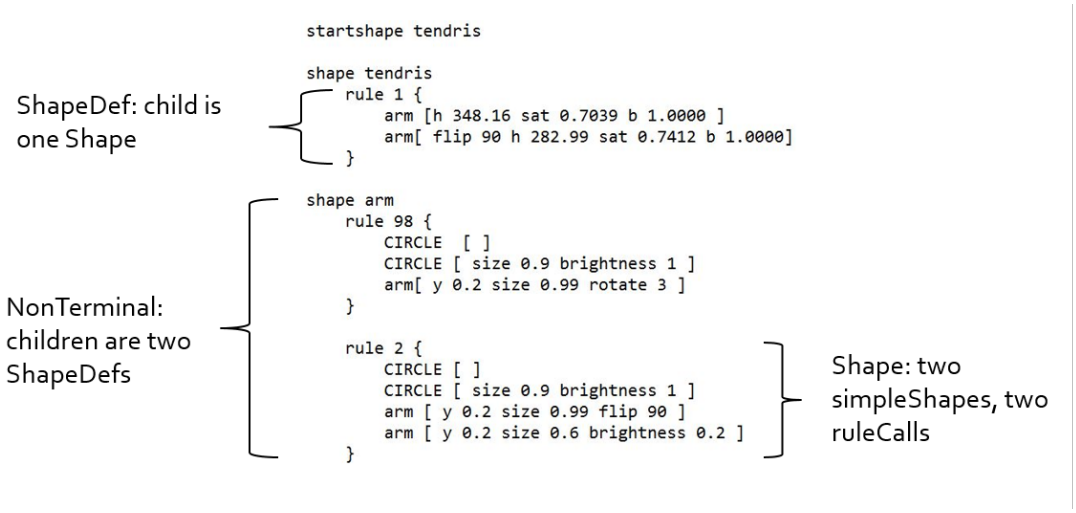

Figure 3.1: Anatomy of a Program

wide, and 10 units tall. Finally, `RuleCalls` are used for calls to user made shapes. This includes both a shape calling itself or other shapes. To prevent infinite recursion, CFA will stop generating shapes once they become too big, or smaller than the declared minimum shape size. Often when generating a program that will look infinitely recursive, there will be a size modifier in the arguments passed to the shape in order for CFA to know when to stop running.

`Program` structure is depicted in Figure 3.2. The lower level components of a `Program` are modeled in Figure 3.3, where each object inherits from the one above it.

3.2 Breeding

What makes a genetic algorithm unique is its ability to create new child programs from parent programs. This is accomplished, in my program, by using crossover and mutation, and then applying it to many children for many generations. Fitness functions are used to try to guide the way that the programs look after multiple generations, ideally in a way that is more interesting.

3.2.1 Crossover

There are three primary functions responsible for crossover: `crossNT`, `crossShapeDef`, and `crossSquences`. `flattenNT` and `slicechildren` are helper functions that are called by `crossNT` and `crossSequences`, respectively.

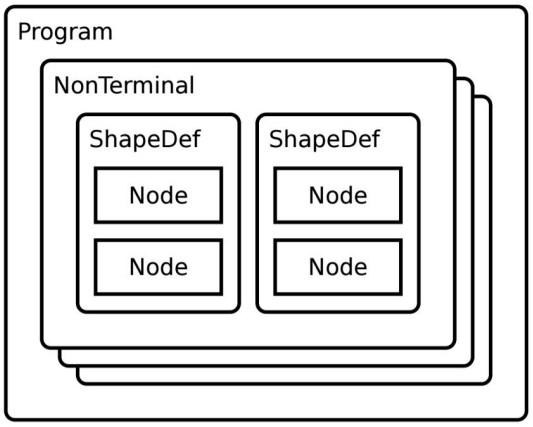

Figure 3.2: Structure of a Program

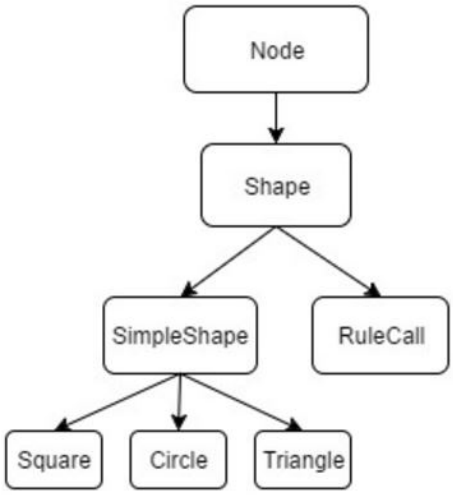

Figure 3.3: Illustration of class inheritance

slicechildren

`slicechildren` takes a list of children of a `ShapeDef`, and a random number. It populates one list, `splitAt`, with a series of positions to split the list of children at. It then fills another list, `toReturn`, with (potentially multiple) children, according to the size of `splitAt`'s indices. Depending on the number of slices made, most of the returned list might be empty. For example, if there is only one child, but `splitAt` has a length of 20, indices 1 - 19 will be empty. It is guaranteed that the first one will not be empty so the program will have something to cross, which means if only one child is put in the list, it will go in index 0.

crossSequences

`crossSequences` divides two "children" of a `ShapeDef` by calling `slicechildren` on each of them with a randomly generated number. Because this is the same number for each list, it will make two lists of the same size populated with grouped parts of the two children. A new list is made that will randomly chose parts from either of these two `slicechildren` lists. This crossover will either take groups from one list or the other, in order to keep sizing as consistent as possible.

crossShapeDef

`crossShapeDef` takes a `ShapeDef` and a "partner" (another `ShapeDef`) to cross with. It will run `crossSequences` on the two, returning a list populated with the results of `crossSequences`. This list is the list of children that the new `ShapeDef` produced by `crossShapeDef` will have. To make sure that there are no problems with this list, each item is checked to make sure it is the right format for what a child of a `ShapeDef` is expected to be. If it is not, that child is deleted. On the off chance that this results in an entirely empty list, the entire process is repeated by recursing on `crossShapeDef`. The weight of this new `ShapeDef` will be either the original or partner's weight. This function is also where mutation occurs; there are three ways this might happen, all described in Section 3.2.2. When the function is complete, it returns a new `ShapeDef` with no parent, a new list of children, and a weight chosen from one of the two crossed `ShapeDefs`.

flattenNT

In order to make sure all of the correct shape rules are called, `flattenNT` goes through the entire program, and makes sure that every shape rule that is called has a corresponding `NonTerminal` for the final image. For example, if the `startshape` calls shape1, which calls shape2, and shape1 is put in the new program after crossover, but shape2 is left behind, `flattenNT` will realize that shape2 is needed for the program to run, and will add it to the new program.

crossNT

`crossNT` is responsible for the crossover of two `NonTerminals`, the highest level at which crossover occurs. It calls `crossSequences` to create a list containing varying amounts of `ShapeDefs` in each index. `crossShapeDef` is then called for each index of the array, picking an appropriate `ShapeDef` to cross with for each one. To add more complexity to the resulting image, `crossNT` adds a 50% chance that an extra crossed `ShapeDef` is added to the image code. The final `NonTerminal` is then "flattened" with `flattenNT`, making sure the final image program runs correctly.

3.2.2 Mutation

There are three ways that mutation might occur in this program, although only one (`mutateParamVal`) is currently in use:

crossParams

`crossParams` takes the list of all of the shape modifications in both of the two programs and has a chance of modifying a parameter to be a random one from this list. The modification would only occur if it is appropriate to do so - if a modifier takes two numbers after it, it would not be swapped with one that takes three.

`crossParams` was meant to be a mutation/crossover hybrid of sorts since it could only be chosen from the parameters present in the two original programs.

The reason this is not currently implemented is that some modifiers are necessary for program success, especially size. If size were switched out for something else, the shape might not ever reach its minimum or maximum size threshold, and never terminate. This is a general problem with genetic programming; at each level of crossover or mutation a potential risk is introduced that a harmful change will occur.

mutateParams

Similar to `crossParams`, `mutateParams` switches the parameter names (e.g. size) of a `Modifier`. This occurs through swapping the parameter name with one in a global list of `Modifiers` that contain the appropriate amount of values afterwards. This means that the mutation would not be restricted to just the parameters found in the two programs, but all of the ones translated from CFA so far. This mutation strategy is also currently disabled due to difficulties with program termination.

mutateParamVal

`mutateParamVal` is the only mutation currently implemented when creating new images. It is called on each parameter of a `Modifier` (the numbers following the `Modifier`), and has

approximately a five percent chance of increasing the number by one percent or decreasing it by one percent.

This does very little from one generation to the next, especially on small values, but allows for more change over multiple generations. This can be detrimental to an image appearance, as a single shape can take over the entire image (see Figure 4.14).

3.2.3 Offspring

Creating multiple offspring is fairly trivial, after the work of creating a single new image program. The function, `programreproduce`, simply takes an empty list of size 100 and fill each index with a new `Program`, generated by `NewProgram`, a function that makes a `Program` with a list of `NonTerminal` children, and the name of the first of these children as its `startshape`. This is done by `NewProgram` calling `crossNT` on the two parent `Programs`, which then calls the rest of the crossover functions, eventually returning a new `Program`.

3.2.4 Multiple Generations

Similarly, creating multiple generations is trivial after creating a single generation. The function, `programbreed`, takes in a list of programs (produced by `programreproduce`), and a number of generations as its input. For each generation, two random `Programs` are picked from the list, and bred together to create another hundred children. The `Programs` are picked randomly, but can be re-chosen if their fitness is undesirable. This is achieved by selecting a child, determining if its fitness is above the average fitness of the 100 children in a generation, and reselecting if the fitness is below average. This process is repeated as many times as needed in order to chose two parents that have an above average fitness.

3.2.5 Fitness

Two fitness functions were tried for this project: the first rewards programs for having more `NonTerminals`, the other rewards programs for having a higher average number of children per `ShapeDef`. To clarify, in the `ShapeDef` fitness function, the children of `ShapeDefs` are `Shapes`, which can either be `RuleCalls` to other `NonTerminals` or `SimpleShapes`, primitives like `Square`, `Circle`, and `Triangle`.

Chapter 4

Evaluation

4.1 Results

Overall, breeding programs together was a success, especially when only looking at a single generation. Figure 4.1 depicts the typical output from parent images that are used throughout all of the experimentation. These are all from Context Free Art's gallery page, created by users of the website [zee] [cra] [Chr] [mom].

Qualitatively, there are four categories that new programs fall into, listed in order of general "interestingness:"

1. Full integration: the two programs merge together such that the individual parent characteristics can be recognized, but they make an image that no longer looks like either of the parents. These images are often the most aesthetically pleasing, as they create something new and interesting.

2. Copy-paste: the image looks like both of the parents, but contains information from each of them. This might mean two full looking programs added together, or a base case of one added to the other.

3. Parent-like: the image has code from both parents, but looks almost identical to a single parent. There are usually some barely-distinguishable differences from parent to child, like slight differences in color or pattern.

4. Base case: the image contains one or a few of the base cases from one of the parents. This results in something such as an image with only a black square on it, for example.

Figure 4.2 contains some of the best products of breeding the parent programs with each other, and themselves. All of the breeding with themselves, seen along the the diagonal, would fall under the "parent" category. Most of the rest are more fully integrated, except for a few copy-paste images, since some parent programs do not breed well together.

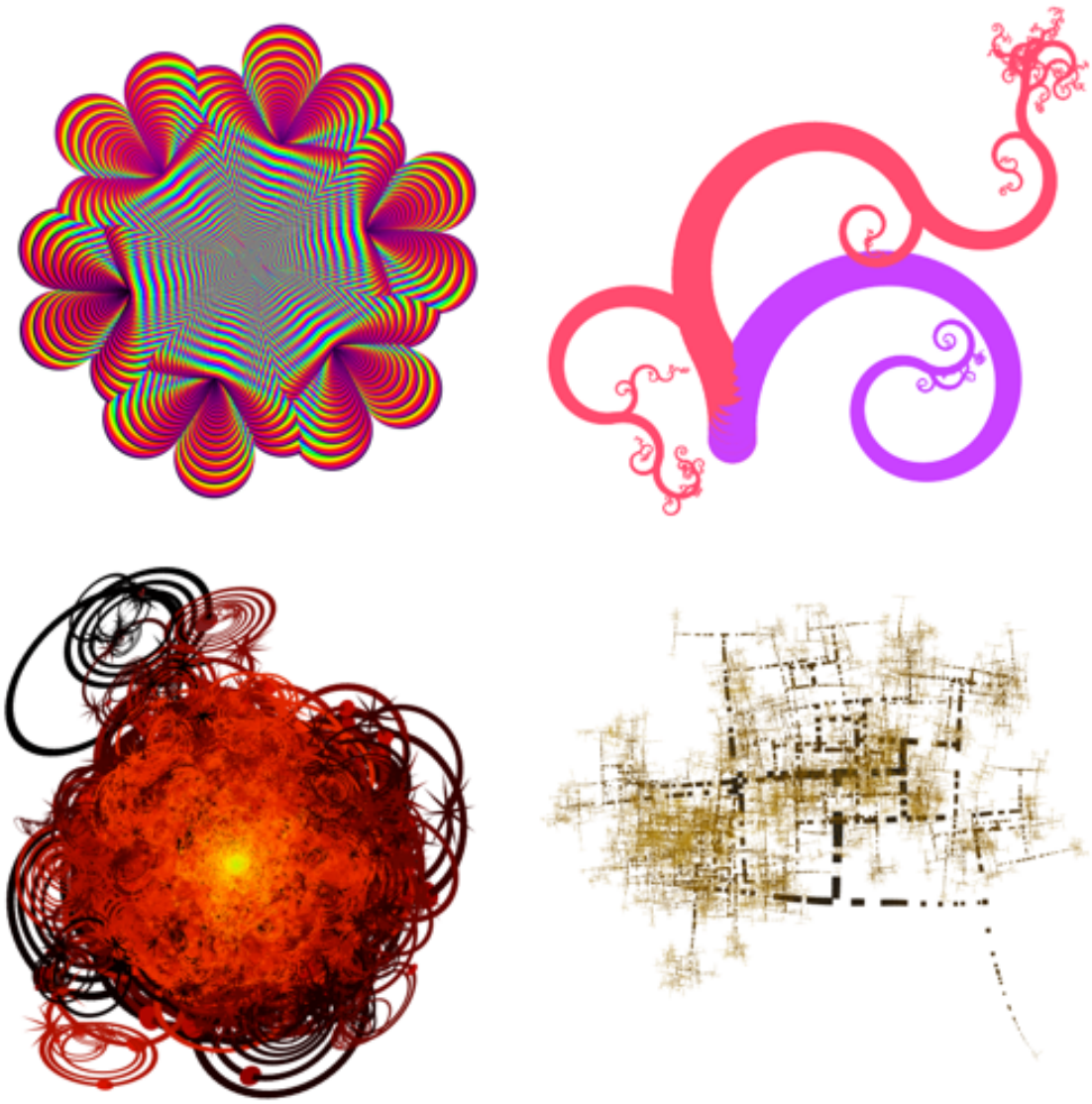

Figure 4.1: Parent images 1-4, read from left to right

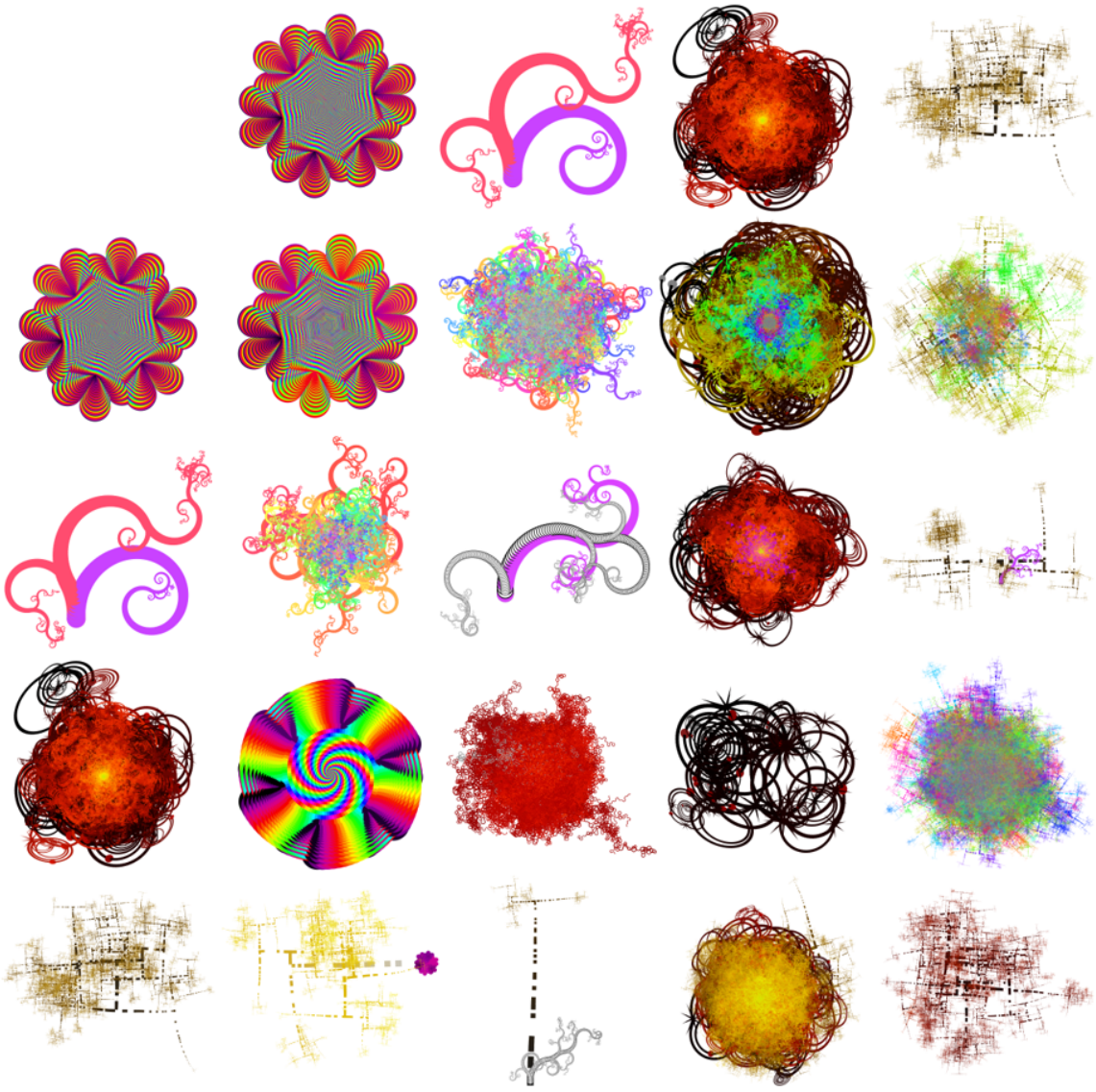

Figure 4.2: Favorites of each parent crossed with the others, itself

Figure 4.3 is produced by breeding parent programs 1 and 2 together nine separate times. These parent programs were selected because, during experimentation, they were more likely to produce integrated images than other programs did. Even with this higher chance of integration, it is still a somewhat rare occurrence. In Figure 4.3, integration only occurs once (outlined in purple), with 4/9 being copy-paste images (outlined in green), 3/9 looking like a parent (outlined in red), and the remaining 1/9 looking like the base case of parent 1 (outlined in blue).

This base case image from Figure 4.3 does not always produce a black flower shape when the code is run. Sometimes it will produce an image that looks like parent 1. Another example of this is depicted in Figure 4.4, where the code will produce a single circle about half of the time, and a much more complicated image the rest of the time.

Figure 4.4's `startshape`, *tendris*, has two rules that it might pick. The first produces a circle, the second produces two circles, and calls *arm*. If the first option is chosen, there is nothing more to be done, and the program terminates, producing a circle. Otherwise it will continue to recurse, probably making many more arms.

This creates a bit of difficulty when assessing the "goodness" of a program, as it is unclear whether each image a program produces will be interesting or not. This could be fixed by running each program multiple times, if desired. So far, the well integrated images do not have this duality, so there has not been a need for producing extra images this way.

4.2 Experimentation with multiple generations

As genetic algorithms parallel nature, they are often run for many generations in order to produce more optimized results. In this case, an optimized program would be one that produces a more interesting image. This was difficult to do, as quantitatively measuring the aesthetics based directly on the program code is not easy. Two fitness functions were used, neither of which made the desired "integrated" programs more often. One explanation for this is that each generation introduces more chaos as they stray from the original parent code; because the next generation only ever contains children of the two parents, there is a rapid divergence from the original code.

To test the viability of each of the fitness functions (none, `Nonterminal` fitness, and `ShapeDef` fitness), parents 1 and 2 were bred to have 100 children, and two children were randomly chosen from these 100 to have the next generation. This was repeated 5, 20, and 100 times to represent 5, 20, and 100 generations of children. Each number of generations was run 9 times so variation could be seen in the output. Figures 4.5 - 4.14 depict the results of varying the fitness, mutation, and number of generations.

By the time a program reaches 100 generations of breeding, the code has become 15-25 times larger than the parent programs (parents 1 and 2 are 12 and 20 lines of code, respectively, including spacing; one of the programs at generation 100 is 304 lines of code). Most of this code, about 286 lines of it, are various options for the `startshape`. This

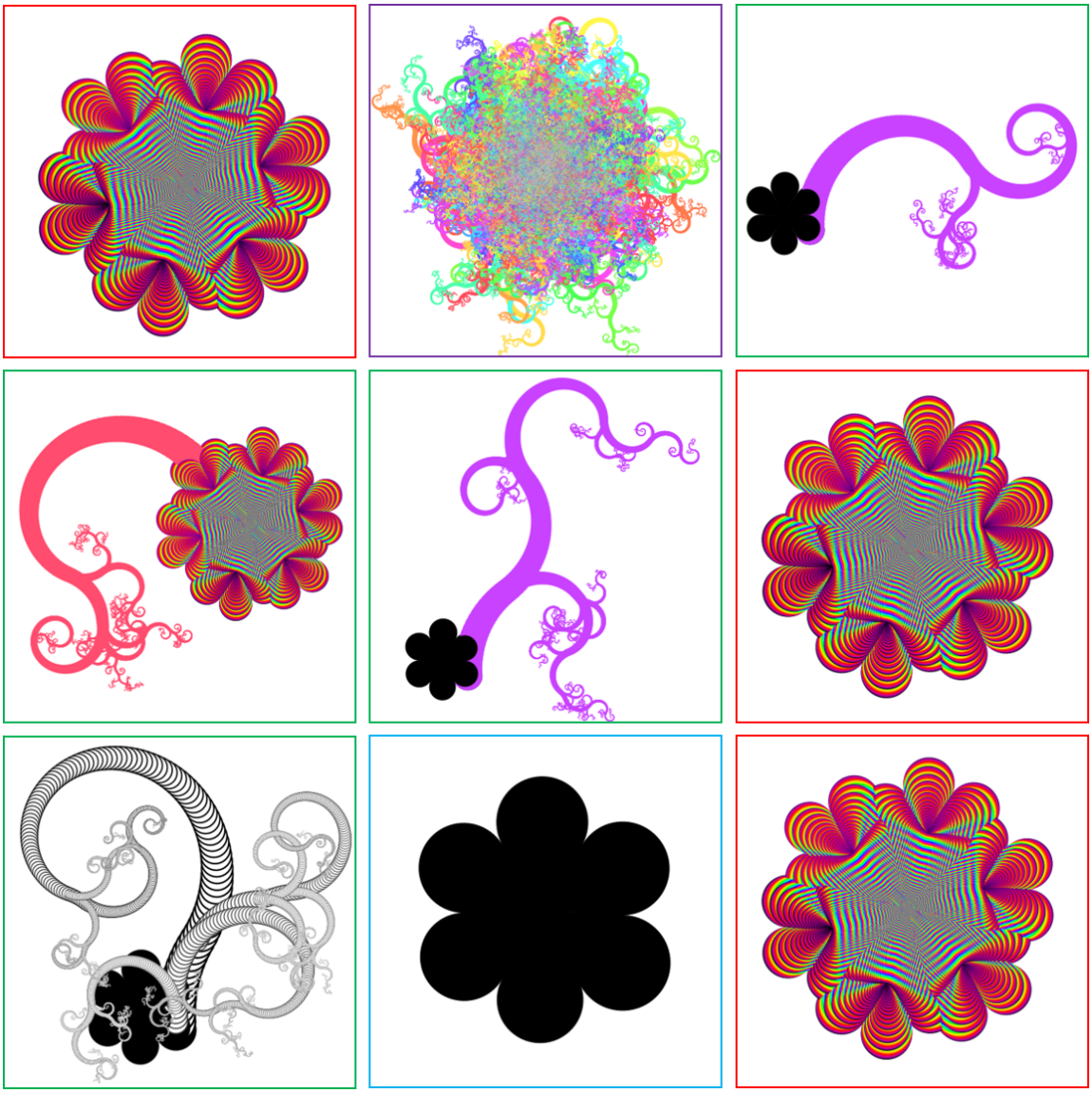

Figure 4.3: Nine images created by combining parents 1 and 2, outlined to show category

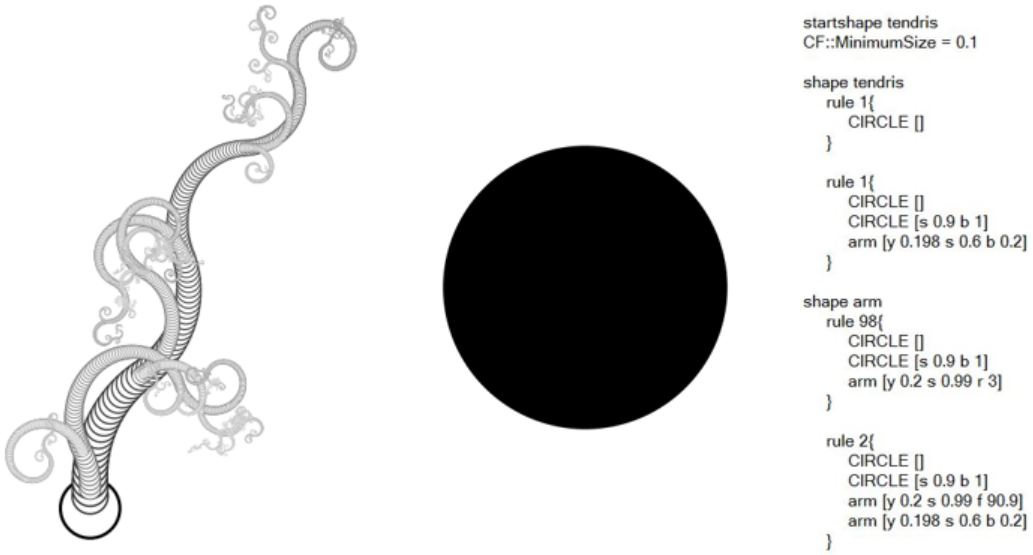

Figure 4.4: Two images produced from the same code (parents 2 and 3)

contributes to much more randomness in later generations of breeding, as there are many more options for the program to choose when generating a shape.

Mutation methods were also tested, to see how they would change the programs over longer periods of time. Two of them were not used because they caused program failure, but one, `mutateParamVal` was tested more thoroughly. `mutateParamVal` does make children look different from their parents, most strikingly by changing the size parameters to the point where many of the produced images looked like a primitive shape. When this occurred, the shapes would often get so large that Context Free would not generate an image file because the program would only terminate because of an error.

`CrossParams` was briefly tested as another form of mutation, but the code crashed the first 8/9 times the program was run, so it was not worth following through with.

4.2.1 Without mutation

In general, not having mutation created many images that look like parent 2, with *tendris* shapes. There are very few images that look unlike their parents; at best they are copy-paste, with many falling closer to the parent or base case categories. There is also very little variety in color, most images are either black and white or one of parent 2's colors. Most of the produced images are ones that would be thrown away when selecting interesting images. Because of this lack of variety, only one example of each round of generations has been included, the other two have been moved to appendix A.

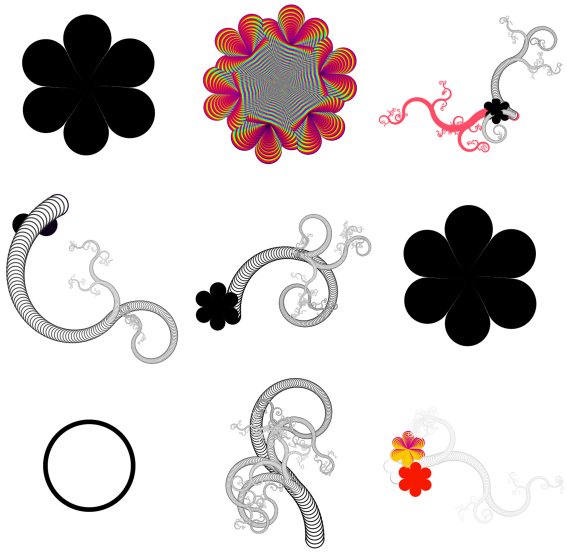

Figure 4.5: Nine images produced from breeding parents 1 and 2 for five generations, without mutation, with NonTerminal fitness

5 generations

After five generations, all figures (Figure 4.5, Figures A.1 and A.2 in appendix A) still have some images that are colored, and produce images that are mostly copy-paste or parent like, with a few base cases in each. It is difficult to say at this point if adding fitness is increasing interestingness.

20 generations

After 20 generations (Figure 4.6, Figures A.3 and A.4 in appendix A), only three out of twenty seven images still has any color. Again, most images fall somewhere in the range of copy-paste to base case. It is difficult to decide because of the small sample size, but at this point, fitness seems to be affecting the programs in a noticeable way, although maybe not desirably. The `NonTerminal` fitness is potentially making more complicated copy-pastes, while the `ShapeDef` fitness is certainly contributing to there being only parent-like images in Figure A.4. Since the `ShapeDef` fitness looks for number of Shapes, it explains the recursive behavior of all of the images in Figure A.4.

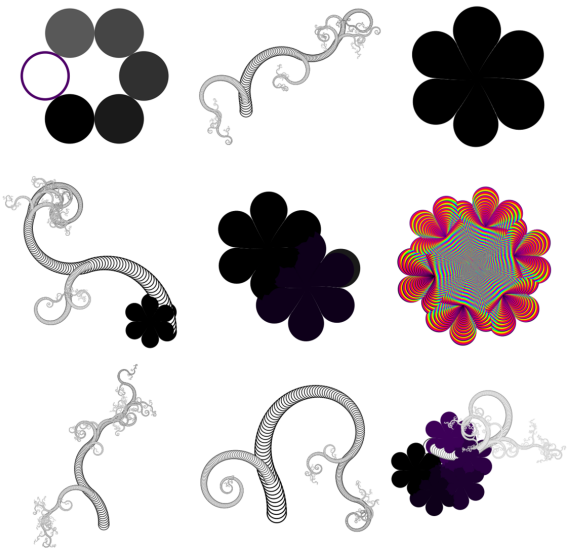

Figure 4.6: Nine images produced from breeding parents 1 and 2 for twenty generations, without mutation, with NonTerminal fitness

100 generations

After 100 generations (Figure 4.7, Figures A.5 and A.6 in appendix A), there seems to be no new developments in images produced. There are images similar to each of the 100 generation images in the 20 generation figures. The fitnesses still might be encouraging complexity and recursion, but not more than they were at 20 generations.

4.2.2 With mutation

Adding mutation certainly changes the general images produced. Most images look almost entirely unlike their parents, especially after more than five generations. As explained in Section 4.2, mutation affected all parameters, including, most importantly, the size. The parents were all written with decreasing sizes, and the mutation would often change at least one to increase instead. This led to programs that were unable to finish, as they would reach a point where the shape size became too big. When this occurred, the images were generated by manually opening context free and saving the image at the point of error.

5 generations

After five generations (Figures 4.8-4.10), relatively new images were being produced, although they were not necessarily more interesting than those that look like their parents. Those produced with no fitness and the `ShapeDef` fitness all looked fairly similar to their

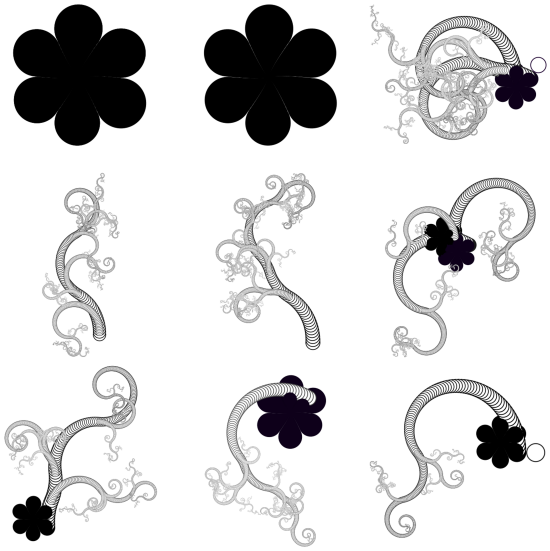

Figure 4.7: Nine images produced from breeding parents 1 and 2 for one hundred generations, without mutation, with NonTerminal fitness

siblings, while those produced by the `NonTerminal` fitness showed more variety. All three versions still have color, and, as with no mutation, fitness does not seem to have a great impact.

20 generations

By 20 generations (Figures 4.11-4.13), many of the images begin to get quite boring. Many look like some form of the base case, which is what is happening: the base case is starting to get so large that it is the main part of the image. This is why so many of the images display tail like attributes, but smaller than those seen with *tendris*. With one exception, all of the images are now black and white. As with before, fitness appears to be affecting the type of shapes being produced, but not in an especially quantifiable way.

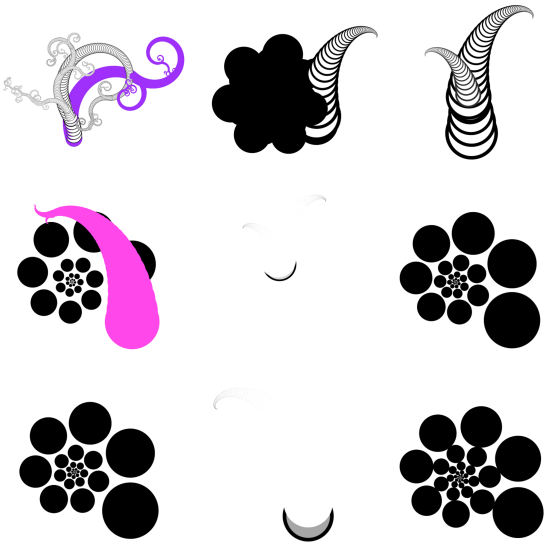

Figure 4.8: Nine images produced from breeding parents 1 and 2 for five generations, with mutation, no fitness

Figure 4.9: Nine images produced from breeding parents 1 and 2 for five generations, with mutation, with NonTerminal fitness

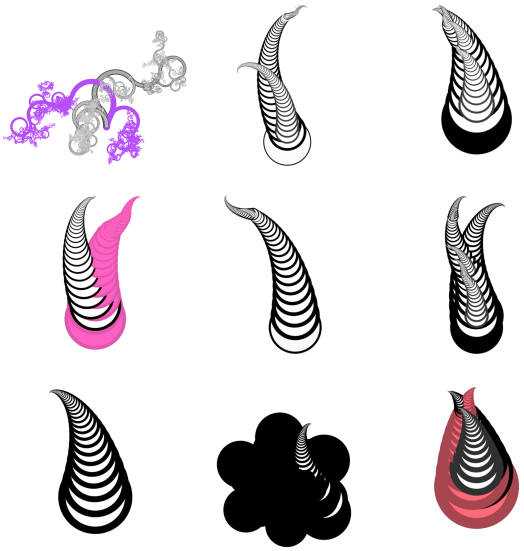

Figure 4.10: Nine images produced from breeding parents 1 and 2 for five generations, with mutation, with ShapeDef fitness

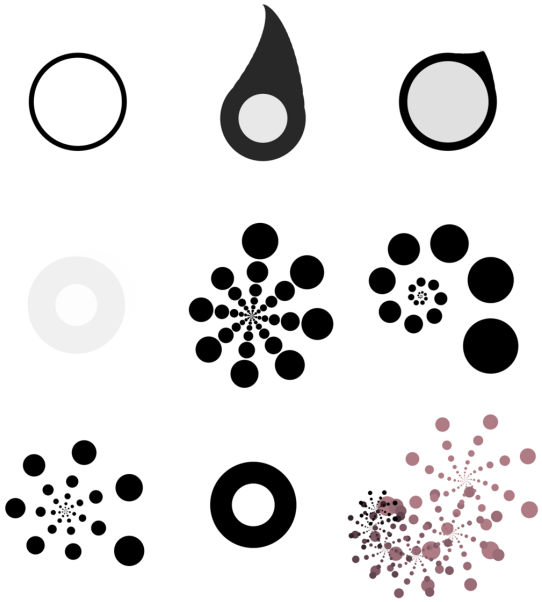

Figure 4.11: Nine images produced from breeding parents 1 and 2 for twenty generations, with mutation, no fitness

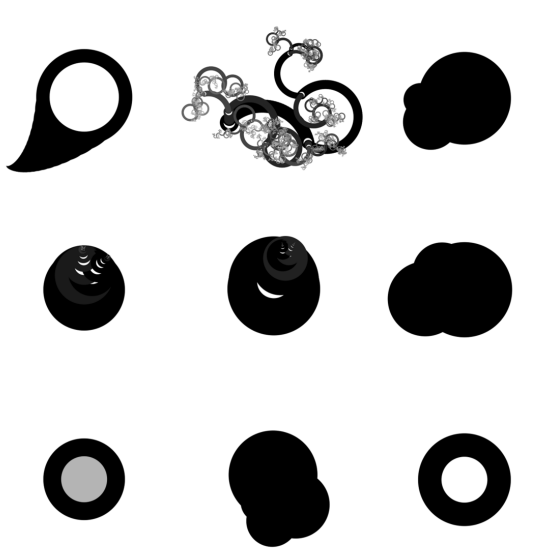

Figure 4.12: Nine images produced from breeding parents 1 and 2 for twenty generations, with mutation, with NonTerminal fitness

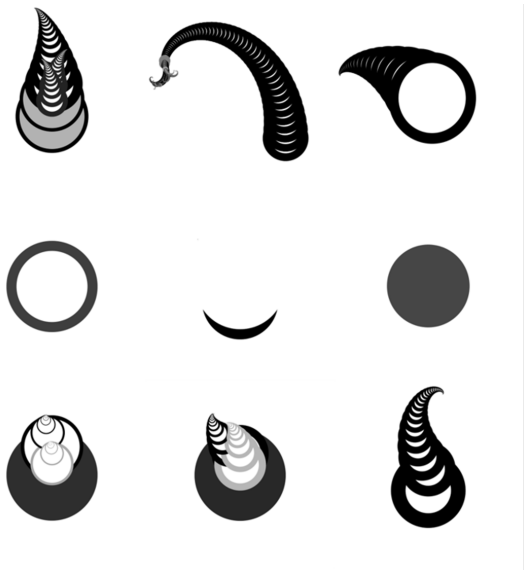

Figure 4.13: Nine images produced from breeding parents 1 and 2 for twenty generations, with mutation, with ShapeDef fitness

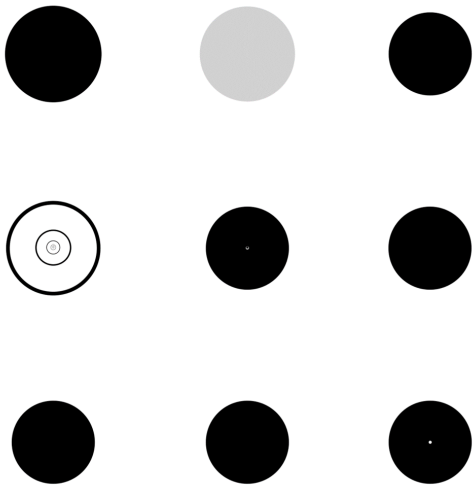

Figure 4.14: Nine images produced from breeding parents 1 and 2 for one hundred generations, with mutation, no fitness

100 generations

By 100 generations (Figure 4.14, Figures A.7 and A.8 in appendix A), the images look almost entirely like the circle base case. The `NonTerminal` fitness, seen in Figure A.7, has the only exception, potentially pointing to an increase of complexity leading to this. By now, all images are black and white, which makes it difficult to notice any additional complexity within them. This is also a problem with the sizing, as the programs generally finish with the largest shape (when they crash), which will often cover all of the other shapes. As with the images without mutation, there is little variety after 100 generations, so two of the figures have been moved to appendix A.

Chapter 5

Conclusions

I set out to create a genetic algorithm designed to read translated Context Free Art programs and produce new, altered programs. This algorithm crosses the two inputs at every level until reaching `Modifiers`, producing a new image that, ideally, is a well combined version of its parents.

The effectiveness of the genetic algorithm was evaluated by examining the images it produced, as this ideal, well combined image is not always created. Because this ideal is not always reached, I outlined the three less interesting cases that might be produced (copy-paste, parent-like, and base case).

In my evaluation, I learned that mutation creates fewer parent-like images, but does not allow for much variety because of the issue with size increase. This is especially true after many generations; after 100 generations, many of the images were difficult to distinguish from one another. I also found that fitness affects the populations' code, but not necessarily in a way that produced more pleasing images.

5.1 Future Work

Overall, it seems that more interesting results are produced after a single generation. Those produced after multiple generations are certainly more complicated, at the code level, but this does not always translate to a more complicated or interesting image. A large contribution to this is the way that breeding happens, as the parent image code is not often preserved over generations. Further experimentation can improve our knowledge of the relationship between program structures and aesthetically pleasing results. If such a relationship exists, the algorithm would be able to optimize its output to make each generation more interesting than the last.

To add to this, further research into an improved mutation function would also be beneficial. This would assure that the the next generations were not only interesting, but also unique. A single program might be interesting, but becomes less so with repetition, as seen with parent-like children.

Increasing the sample size, both through trial breeding and more diversity in parents, would aid in determining what makes a program interesting. Stronger analysis of the programs and output would be needed, to evaluate the way that fitness, mutation, and multiple generations are affecting the outputted images.

Finally, there is a limitation to coding only Context Free Art programs that can be translated. Some aspects of Context Free Design Grammar have not yet been included, the addition of which would allow for more complex programs, and more variety in offspring. Making any Context Free Art program translatable and able to be bred would allow for potentially more interesting child programs to be created.

Appendices

Appendix A

Additional Images

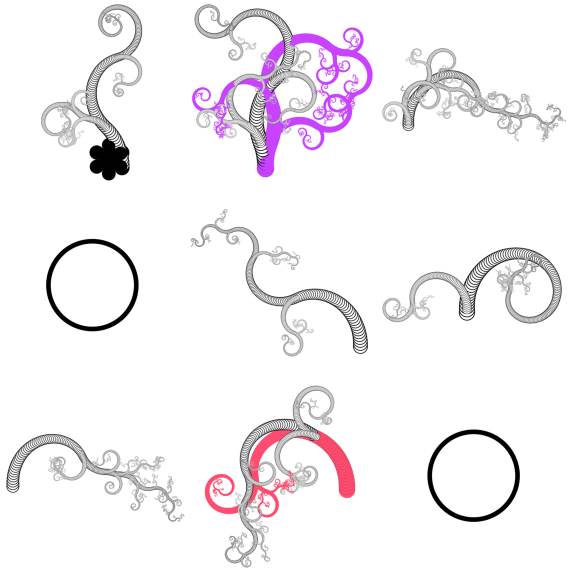

Figure A.1: Nine images produced from breeding parents 1 and 2 for five generations, without mutation or fitness

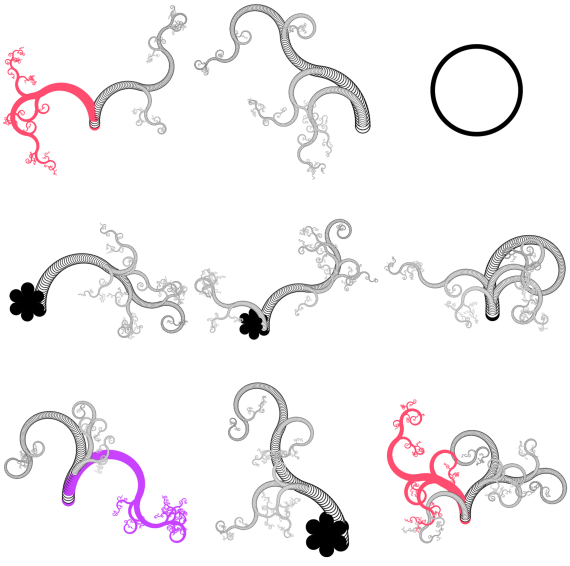

Figure A.2: Nine images produced from breeding parents 1 and 2 for five generations, without mutation, with ShapeDef fitness

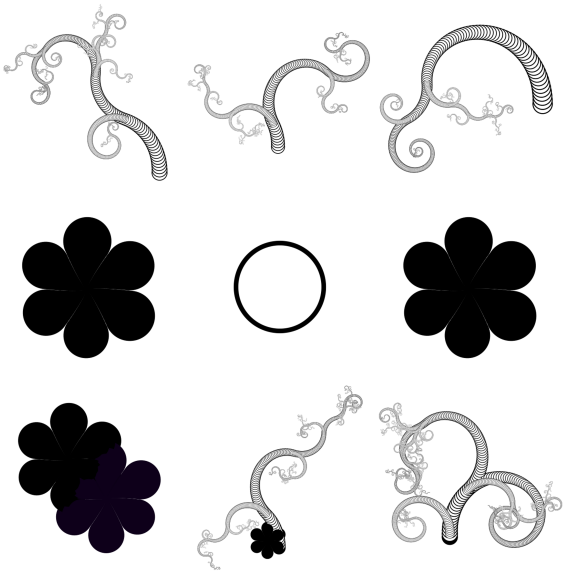

Figure A.3: Nine images produced from breeding parents 1 and 2 for twenty generations, without mutation or fitness

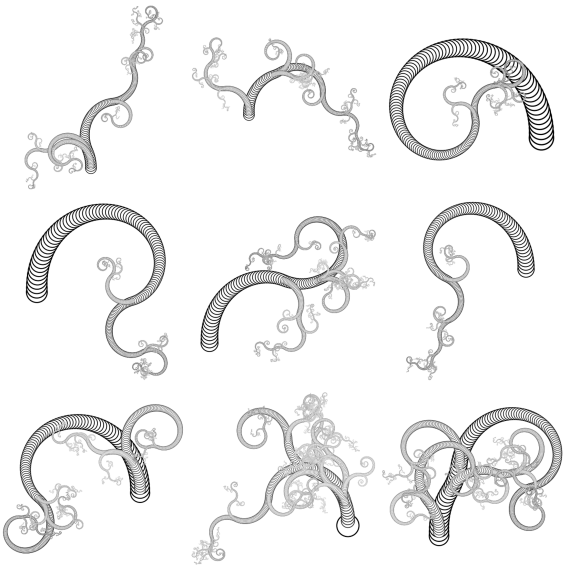

Figure A.4: Nine images produced from breeding parents 1 and 2 for twenty generations, without mutation, with ShapeDef fitness

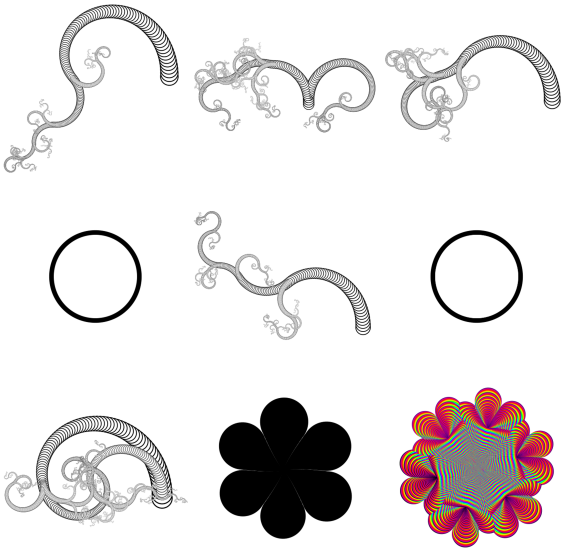

Figure A.5: Nine images produced from breeding parents 1 and 2 for one hundred generations, without mutation or fitness

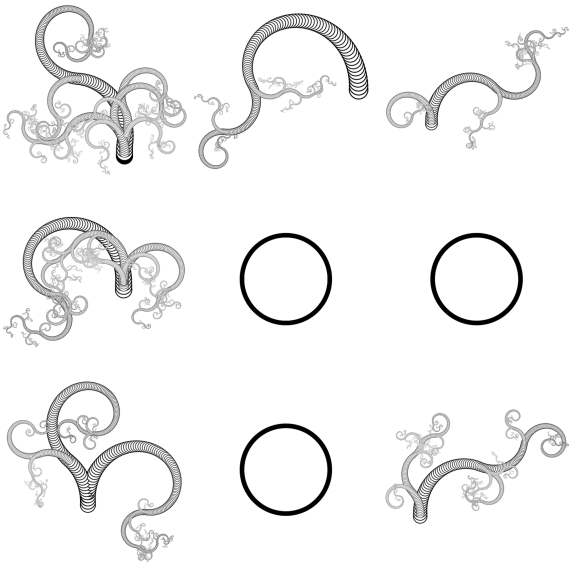

Figure A.6: Nine images produced from breeding parents 1 and 2 for one hundred generations, without mutation, with ShapeDef fitness

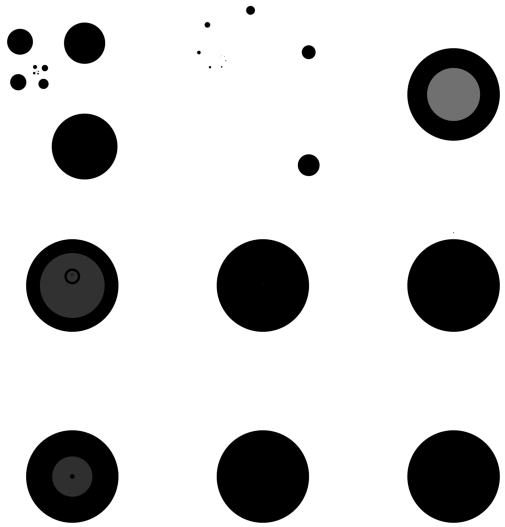

Figure A.7: Nine images produced from breeding parents 1 and 2 for one hundred generations, with mutation, with NonTerminal fitness

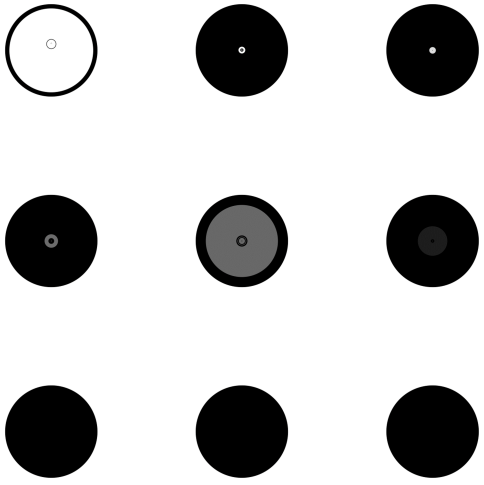

Figure A.8: Nine images produced from breeding parents 1 and 2 for one hundred generations, with mutation, with ShapeDef fitness

Appendix B

Code

Note: this code can also be accessed on GitHub at *https://github.com/mtkent/SeniorProject*

```
# classes.py: contains all class definitions for program

class Program: # has startshape, which shapeDef, and a dictionary
    of nodes
        def __str__(self):
                return """
                startshape {startshape}
                CF::MinimumSize = 0.1
                {shapes}
                """.format(startshape = self.startshape.name,
                    shapes = "\n".join(str(k) for k in self.shapes
                    ))
        def __init__(self, startshape, shapes): # shapes should
            be dictionary
                self.shapes = shapes # add if statment for
                    dictionary
                self.startshape = self.findNT(startshape)
                if self.startshape == None:
                        print ("shape ", startshape, "not found
                            in NonTerminals")
        def findNT (self, name):
                for c in self.shapes:
                        if (c.name == name):
                                return c
                return None
        def addShape (self, shape):
```

```
                self.shapes[shape.name] = shape

class NonTerminal:
        def __str__(self):
                return ("shape " + self.name + "\n" + "\n".join(
                    str(k) for k in self.children))
        def __init__ (self, name, children, program = None):
                self.children = list(children)
                self.name = name
                for c in self.children:
                        c.parent = self
                self.program = program
        def setProgram (self, p):
                self.program = p
        def addShapeDef (self, shapedef):
                self.children.append(shapedef)
                shapedef.parent = self
        def __copy__ (self):
                return self.copyHelper({})
        def copyHelper (self, dictionary):
                if self in dictionary:
                        return dictionary[self]
                else:
                        clist = []
                        result= NonTerminal(self.name, [])
                        dictionary[self] = result
                        for c in self.children:
                                result.addShapeDef(c.copyHelper(
                                    dictionary))
                        return result

class ShapeDef:
        def __str__(self):
                return "rule " + str(self.weight) + "{\n" + "\n".
                    join(str(x) for x in self.children) +  " \n }
                    \n"
        def __init__ (self, parent, children, weight = 1):
                self.parent = parent
                self.children = children
                self.weight = weight
        def __copy__ (self):
```

```
                return self.copyHelper({})
        def copyHelper (self, dictionary):
                if self in dictionary:
                        return dictionary[self]
                else:
                        cList = []
                        result = ShapeDef(None, cList, self.
                           weight)
                        dictionary[self] = result
                        for c in self.children:
                                cList.append(c.copyHelper(
                                   dictionary))
                        return result

class Node:
        def __str__(self):
                return "unimplemented"
        def __init__(self, children):
                self.children = children
        def __copy__ (self):
                return self.copyHelper({})
        def copyHelper (self, dictionary):
                if self in dictionary:
                        return dictionary[self]
                else:
                        cList = []
                        result = type(self)(cList)
                        dictionary[self] = result
                        for c in self.children:
                                cList.append(c.copyHelper(
                                   dictionary))
                        return result

class Shape(Node):
        def __str__(self):
                return "{} [{}]".format(self.name, self.argsStr()
                    )
        def __init__(self, name, args):
                super().__init__(args)
                self.name = name
        def argsStr(self):
```

```
                return "␣".join(str(c)for c in self.children)

class SimpleShape (Shape):
        def __init__ (self, args):
                super().__init__(None, args)
                del self.name

class RuleCall (Shape):
        def __init__(self, rule, args):
                if (rule == None):
                        super().__init__("__noName__", args)
                else:
                        super().__init__(rule.name, args)
                self.rule = rule
        def __str__(self):
                self.name = self.rule.name
                return super().__str__()
        def setRule (self, rule):
                self.rule = rule
                self.name = rule.name
        def __copy__ (self):
                return self.copyHelper({})
        def copyHelper (self, dictionary):
                if self in dictionary:
                        return dictionary[self]
                else:
                        cList = []
                        result = type(self)(None, cList)
                        dictionary[self] = result
                        toPrint = self.rule.copyHelper(dictionary
                           )
                        result.setRule(toPrint)
                        for c in self.children:
                                cList.append(c.copyHelper(
                                   dictionary))

                        return result

# shapes
class Square(SimpleShape):
        name = "SQUARE"
```

```
class Circle(SimpleShape):
        name = "CIRCLE"

class Triangle(SimpleShape):
        name = "TRIANGLE"

class Modifier:
        def __str__(self):
                return "{} {}".format(self.name, " ".join(str(v)
                   for v in self.values))
        def __init__(self, *values):
                self.values = values
        def __copy__ (self):
                return self.copyHelper({})
        def copyHelper (self, dictionary):
                if self in dictionary:
                        return dictionary[self]
                else:
                        return type(self)(*self.values)

# modifers and value
# 1
class Alpha(Modifier):
        name = "a"
# 1
class Brightness(Modifier):
        name = "b"
# 1
class Saturation(Modifier):
        name = "sat"
# 1
class Hue(Modifier):
        name = "h"
# 1
class Y(Modifier):
        name ="y"
# 1
class Z(Modifier):
        name ="z"
# 1
```

```
class Rotate(Modifier):
        name = "r"
# 1
class Flip(Modifier):
        name = "f"
# 1, 2, 3
class X(Modifier):
        name = "x"
# 1, 2, 3
class Size(Modifier):    # if this larger than certain number,
    does not work well
        name = "s"

# takes 1, 2, 4, 6
class Transform(Modifier):
        name = "trans"

# takes 2
class Skew(Modifier):
        name = "skew"

# takes 1
class randRange(Modifier):
        name = ".."

class Value:
        def __str__(self):
                return repr(self.val)
        def __init__(self, val):
                self.val = val

# functions.py: contains all functions to be used for generating
    new programs

# imports
import math
import random
import subprocess
import os
import copy
import string
```

```
import classes
from classes import Triangle, Square, Circle, Skew, Alpha,
    Brightness, Saturation, Hue, Y, Z, Rotate, Flip, X, Transform,
     ShapeDef, NonTerminal, Shape, Program, RuleCall, Modifier

# lists of all modifiers - not currently used
# mod1 = ["a", "b", "sat", "h", "y", "z", "r", "f", "x"]
# mod2 = ["s", "skew"]

# will pull needed code to make sure it calls everything it
    should
def flattenNT (nt, soFar = None):
        if soFar == None:
                soFar = {nt}
        result = [nt]

        for rule in nt.children:
                for child in rule.children:
                        if isinstance(child, RuleCall) and child
                           not in result:
                                if child.rule not in soFar:
                                        soFar.add(child.rule)
                                        result.extend(flattenNT((
                                           child.rule), soFar))
        return result

def pickPartner (rule, p1, p2):      # finding a suitable match -
    for shapedefs
        parent = rule.parent.program
        otherParent = (p1, p2) [parent == p1]  # clever tuple
           work

        ran = random.choice(otherParent.shapes)

        return random.choice(ran.children)

# make a program into something of size 1, with each line taking
    a certain amount of space
# program has a name and a list of shapes - startshape and shapes
# working fairly well, will sometimes drop ones, especially at
    end?
```

```
def slicechildren (children, numparts):
        toReturn = [None] * numparts
        size = int(math.ceil(len(children) / numparts))
        splitAt = [0]
        for i in range (numparts):
                # has a list of numbers to split at
                splitAt.append(splitAt[-1] + size)

        for i in range (numparts):

                # old way of doing this:

                # ran = random.uniform(0,99)
                # # the lower the number the fewer shapes will go
                    into final program
                # # have higher chance of overlap - good for
                    small programs, but LOTS of repetition, since
                    breeding takes away varience
                # if (ran < 75):
                #       size1 = size + 1
                # else:
                #       size1 = size - 1

                # start = i * size

                # will add to toReturn the index from one split
                    to another
                toReturn[i] = children[splitAt[i]: splitAt[i +
                    1]]

        return toReturn

paramArr = []

# sort of mutation/crossover hybrid
def crossParams (param):
        ran = random.uniform(0, 99)
        if ran < 30:
                return random.choice(paramArr)
        else:
                return param
```

```
# picks a random param
def mutateParams (param):  # disabled
        # ran = random.uniform(0, 99)
        toReturn = param
        # if param in mod1:
        #       if ran < 3:
        #               toReturn = random.choice(mod1)

        # elif param in mod2:
        #       if ran < 3:
        #               toReturn = random.choice(mod2)

        return toReturn

# increase or decrease by a percent - not always good for
    multiple generations
def mutateParamVal (param):
        ran = random.uniform(0, 99)
        if ran > 94:
                return param * 1.01
        if ran < 5:
                return param * 0.99
        else:
                return param

# knows how to cross shapes - a single one at a time
def crossSequences (s1, s2):
        splits = [3, 4, 5, 8, 34]
        numparts = random.choice(splits)
        p1arr = slicechildren(s1, numparts)
        p2arr = slicechildren(s2, numparts)

        attributes = []
        for i in range (0, numparts):
                ran = random.uniform(0,99)

                if (ran < 50):
                        attributes.extend(p1arr[i])
                        if len(p2arr[i]) > 0:
                                for j in range(len(p2arr[i])):
```

```
                                        if not (j == 1 or j == 0)
                                            :
                                                paramArr.extend([
                                                    j])
                else:
                        attributes.extend(p2arr[i])
                        if len(p1arr[i]) > 0:
                                for j in range(len(p1arr[i])):
                                        if not (j == 1 or j == 0)
                                            :
                                                paramArr.extend([
                                                    j])
        return attributes

# crossing the shapedefs - will cross its children: (simple)
    shapes
def crossShapeDef(rule, partner, p1, p2):   #add extra rule -
    more complexity
        result = []
        rprog = rule.parent.program
        pprog = partner.parent.program
        children = rule.children
        crosschildren = crossSequences(rule.children, partner.
            children)

        # call cross attributes

        weight = random.choice([rule.weight, partner.weight])
        lenVar = len(crosschildren)
        n = 0

# makes sure don't have double lists - maybe no longer needed
        for c in range(lenVar):
                for n in range(lenVar):
                        if n < lenVar:
                                if lenVar > 0:
                                        for i in crosschildren[n
                                            ].children:
                                                if (isinstance(i,
                                                    list)):
```

```
                                        crosschildren
                                            .
                                            remove
                                            (
                                            crosschildren
                                            [n])
                                        lenVar -=
                                             1
                                        n = 0
                                else:
                                        paramArr.
                                            extend
                                            ([i])
                                            # all
                                            of the

                                            current

                                            params
                                        n += 1

    if (len(crosschildren) == 0):
            return crossShapeDef(rule, partner, p1, p2)

    # good place for mutation?
    for c in range(len(crosschildren)):
            newChildren = []
            for p in crosschildren[c].children:
                    old = p
                    # p = crossParams(p)                    #
                        don't do this.
                    # newName = mutateParams(p.name)
                    newValues = []
                    for val in p.values:
                            newValues.append (mutateParamVal(
                                val))       # param mutation

                    # p.name = newName
                    # p.values = newValues
                                                          #
                        param mutation "switch"
```

```
                        newChildren.extend([p])
                        crosschildren[c].children = newChildren

        return ShapeDef(None, crosschildren, weight)

# crosses nonterminals, which calls the crossing of its children:
    shapedefs
def crossNT (nt1, nt2, p1, p2):
        rules = crossSequences(nt1.children, nt2.children)
        result = []

        for rule in rules:
                partner = pickPartner (rule, p1, p2) # a shapedef
                newShapeDef = crossShapeDef(rule, partner, p1, p2
                    )
                result.append(newShapeDef)

        ran = random.uniform(0,99)

        # 50 percent chance of an extra shapedef
        if ran > 50:
                rule = random.choice(rules)
                partner = pickPartner (rule, p1, p2) # a shapedef
                newShapeDef = crossShapeDef(rule, partner, p1, p2
                    )
                result.append(newShapeDef)

                name = nt1.name
        else:
                name = nt2.name

        returnNT = NonTerminal(name, result)
        return flattenNT(returnNT)

# not implemented - used if programs have same names. Maybe
    currently buggy?
def scramblenames (nts):
# add unique prefixes to names of program - programs don't have
    names...
        i = 0
        for nt in nts:
```

```
                nt.name += str(i) # + nt.name(:20)
                i += 1

# crossover for programs - doesn't take care of shape parameters
# will need to iterate if more than one shape
def newprogram (p1, p2):
        result = crossNT(p1.startshape, p2.startshape, p1, p2)

        # scramblenames(result)                                  #does
            this mean we won't call any shapes correctly?
        return (Program(result[0].name, result))

# breeding and reproduction - creates 100 children
def programreproduce(arr, prog1, prog2):
        for i in range(100):
                child = newprogram(prog1, prog2)
                arr[i] = child

def fitness (program):
        # new fitness:

        # want to determine the size of the program
        # num = len(program.shapes)
        # sum = 0
        # total = 0
        # for i in range (num):
        #       children = program.shapes[i].children   #
            shapedefs
        #       for j in range(len(children)):
        #               sum += len(program.shapes[i].children[j].
            children) # children of a shapedef: shapes: rules or
            simpleshapes
        #                total += 1

        # avg = sum/total
        # return avg  # number of things within rule

        # old fitness:
        return len(program.shapes) # num nonterminals

 #  get average for a program
```

```
def avgFitness (parr):
        total = 0
        num = len (parr)
        for i in range (num):
                total += fitness(parr[i])

        return (total/num)

# make sure good fitness, pick a program
def pickProgram (programarr, num):
        avg = avgFitness(programarr)
        p1 = programarr[num]

        ran = int(random.uniform(0,99))

        if (fitness(p1) < avg):
                p1 = pickProgram(programarr, ran)

        return p1

# makes num generations from 100 children
def programbreed (programarr, num):

        for _ in range(num):

                ran = int(random.uniform(0,99))
                rand = int(random.uniform(0,99))

                p1 = pickProgram(programarr, ran)       # adding
                    fitness increases avg, definitely, but does
                    not mean better pic
                p2 = pickProgram(programarr, rand)

                programreproduce(programarr, p1, p2)

# make a cfa image
def createImage(code, codeName, resultName):
        if (not os.path.exists("output")):
                os.mkdir("output")

        with open("output/" + codeName + ".cfdg", "w") as fout:
```

```
                fout.write(code)

        subprocess.run(["ContextFree/ContextFreeCLI.exe", "output
            /" + codeName + ".cfdg", "output/" + resultName + ".
            png"])

# testing.py: where all testing and program creation happens

# imports
import classes
from classes import Triangle, Size, Square, Circle, Skew, Alpha,
    Brightness, Saturation, Hue, Y, Z, Rotate, Flip, X, Transform,
     ShapeDef, NonTerminal, Shape, Program, RuleCall, randRange
import functions
from functions import newprogram, createImage, programbreed,
    programreproduce, scramblenames, crossParams, fitness,
    avgFitness


# ------------------------------ parent definition here
    ---------------------------------------


# starter parents
parent1 = Triangle([Skew(20, 30), Brightness(.5)] ) #, Hue(41312)
    , Y(100))
parent2 = Square([Transform(45, 100), Flip(5)]) #, Alpha(3),
    Saturation(44))

# more complicated parents
parent3 = ShapeDef(None, [
        Square([Transform(-3, -3)]),
        Square([Transform(3, 3)]),
        Square([Transform(3, -3)]),
        Square([Transform(-3, 3)]) ])

nt5 = NonTerminal("nt5", [])


parent4 = ShapeDef(None, [
        Triangle([Y (10)]),
        Triangle([Y (5)]),
```

```
        Triangle([Y (0)]),
        RuleCall(nt5,[Size (0.90)])
        ])
nt4 = NonTerminal("nt4", [parent4])  # <- adding another parent
    just adds it to it as a rule

parent5 = ShapeDef(None, [
        Circle ([X (2), Skew (12, 45), Hue (45), Rotate (33) ]),
        Square([Transform(3, 3), Hue (12)]),
        Square([Transform(3, 13), Hue (12)]),
        Triangle([Saturation (300), Alpha (43), Transform(4)]),
        RuleCall(nt4, [])

        # Shape(nt4.__copy__(), parent4)
        ])

nt5.addShapeDef(parent5)


# first recursive parent
blahargs = [
        Triangle ([]),
        ]


terminatingShape = ShapeDef (None, [
        Circle([])
        ])

blah2shape = ShapeDef(None, blahargs)
blah2 = NonTerminal("blah2", [blah2shape])


terminatingShape.weight = 0.1

blahargs.append(RuleCall(blah2, [Alpha (0.04), Rotate (47.83),
    randRange(10), X (1), Size (0.9995), Saturation(0.7)]))

blahshape = ShapeDef(None, [
        RuleCall(blah2, [Alpha (-1)]) ,
```

```
        RuleCall(blah2, [Flip (163), Alpha (-1), X (1),
            Brightness (1)]),
        RuleCall(blah2, [Alpha (-10), Y (-5), Brightness(1)]),
        RuleCall(blah2, [Flip (163), Alpha (-1), X (5), Y (-5)])
        ])


nt6 = NonTerminal("blah", [blahshape])

program7 = Program ("blah", [nt6, blah2])

nt6.setProgram(program7)

# nonterminals from more complicated parents
nt3 = NonTerminal("nt3", [parent3])
# nt4 = NonTerminal("nt4", parent4)  #<- adding another parent
    just adds it to it as a rule
# nt41 = NonTerminal("nt41", parent4)
nt5 = NonTerminal("nt5", [parent5])
# nt6 = NonTerminal("nt6", blah2)
# nt7 = NonTerminal("nt7", blah)

# programs from nonterminals
program1 = Program("nt3", [nt3])
program2 = Program("nt4",[nt4, nt5])
program3 = Program("nt5",[nt5, nt4.__copy__()])

# need to set parents after
nt3.setProgram(program1)
nt4.setProgram(program2)
nt5.setProgram(program3)


# tendris parent: https://contextfreeart.org/gallery/view.php?id
    =3807

armArgs1 = [
        Circle ([]),
        ]

armArgs2 = [
```

```
        Circle ([]),
        ]

armShape1 = ShapeDef(None, armArgs1, 98)
armShape2 = ShapeDef(None, armArgs2, 2)

arm = NonTerminal("arm", [armShape1, armShape2])
armArgs1.append(Circle([Size (0.9), Brightness (1)]))
armArgs1.append(RuleCall (arm, [Y (0.2), Size (0.99), Rotate (3)
    ]))


armArgs2.append(
        Circle([Size (0.9), Brightness (1)])
        )
armArgs2.append(RuleCall (arm, [Y (0.2), Size (0.99), Flip (90)])
    )
armArgs2.append(RuleCall (arm, [Y (0.2), Size (0.6), Brightness
    (0.2)]))

tendrisShape = ShapeDef (None, [
        RuleCall (arm, [Hue (348.16), Saturation (0.7039),
            Brightness (1.0000)]),
        RuleCall (arm, [Flip (90), Hue (282.99), Saturation
            (0.7412), Brightness (1)])
])


tendris = NonTerminal("tendris", [tendrisShape])

online1 = Program("tendris", [tendris, arm])

tendris.setProgram(online1)
arm.setProgram(online1)



# flower parent: https://contextfreeart.org/gallery/view.php?id
    =122
cs1args = [
        Square([])
```

```
]
cs2args = [
        Circle ([Size (3.5), Brightness (0.5)])
]
cs3args = [
        Square([])
]

flowerArgs = [
        Triangle([Size (15, 1), Rotate (45)])
]
flowerShape = ShapeDef(None, flowerArgs)
flower = NonTerminal("flower", [flowerShape])
flowerArgs.append(RuleCall(flower, [Size (0.9), Rotate(45)]))

startArgs = []
sceneArgs = []

curveShape1 = ShapeDef(None, cs1args, 1)
curveShape2 = ShapeDef(None, cs2args, 0.007)
curveShape3 = ShapeDef(None, cs3args, 0.01)

sceneShape = ShapeDef(None, sceneArgs)
startShape = ShapeDef(None, startArgs)
start = NonTerminal("start", [startShape])


curve = NonTerminal("curve", [curveShape1, curveShape2,
   curveShape3])
cs1args.append(RuleCall(curve, [Y (1), Size (0.997), Rotate (5)])
   )
cs2args.append(RuleCall(curve, [Y (1), Size (0.99), Rotate (10)])
   )
cs3args.append(RuleCall(flower, []))
cs3args.append(RuleCall(curve, [Y (1), Size (0.99), Rotate (-40),
    Skew (10, 0)]))

scene = NonTerminal("scene", [sceneShape])
sceneArgs.append(RuleCall(curve, []))
sceneArgs.append(RuleCall(start, [Size (0.995), Rotate (20),
   Brightness (0.01), Hue (0.1), Saturation (0.8)]))
```

```
startArgs.append(RuleCall(scene, [Brightness (0.01), Hue (0),
    Saturation (0.8)]))

online2 = Program("start", [start, scene, curve, flower])
curve.setProgram(online2)
scene.setProgram(online2)
flower.setProgram(online2)
start.setProgram(online2)



# map parent: https://contextfreeart.org/gallery/view.php?id=185

wsArg1 = []
wsArg2 = [
        Square ([])
]
wsArg3 = [
        Square([])
]
wsArg4 = []

wallShape1 = ShapeDef (None, wsArg1)
wallShape2 = ShapeDef (None, wsArg2)
wallShape3 = ShapeDef (None, wsArg3, 0.09)
wallShape4 = ShapeDef (None, wsArg4, 0.005)

wall = NonTerminal("wall", [wallShape1, wallShape2, wallShape3,
    wallShape4])

wsArg1.append(RuleCall(wall, [Y (0.95), Rotate (1), Size (0.975)
    ]))
wsArg2.append(RuleCall(wall, [Y (0.95), Rotate (-1), Size (0.975)
    , Saturation (0.1), Brightness (0.01), Hue (0.1)]))
wsArg3.append(RuleCall(wall, [Y (0.95), Rotate (90), Size (0.975)
    ]))
wsArg3.append(RuleCall(wall, [Y (0.95), Rotate (-90), Size
    (0.975)]))
wsArg4.append(RuleCall(wall, [Y (0.97), Rotate (90), Size (1.5)])
    )
```

```
wsArg4.append(RuleCall(wall, [Y (0.97), Rotate (-90), Size (1.5)
    ]))

ancientmapShape = ShapeDef(None, [
        RuleCall(wall, [Brightness (0.1), Hue (34)]),
        RuleCall(wall, [Brightness (0.1), Rotate (180), Hue (34)
            ])
        ])
ancientmap = NonTerminal("ancientmap", [ancientmapShape])

online4 = Program("ancientmap", [ancientmap, wall])
ancientmap.setProgram(online4)
wall.setProgram(online4)



# sun parent: https://contextfreeart.org/gallery/view.php?id=1872
sunShapeArgs = []
cordShapeArgs = [
        Circle([Saturation (1), Hue (270)])
]
sunShape = ShapeDef(None, sunShapeArgs)

cordShape = ShapeDef(None, cordShapeArgs)


sun = NonTerminal("sun", [sunShape])
cord = NonTerminal("cord", [cordShape])

sunShapeArgs.append(RuleCall(cord, []))
sunShapeArgs.append(RuleCall(sun, [X (1), Rotate (60), Hue (3),
    Saturation (-0.19), Size (0.999), Brightness (0.1)]))
cordShapeArgs.append(RuleCall(cord, [Y (1), Rotate (60.1), Size
    (0.98)]))

online5 = Program ("sun", [sun, cord])
sun.setProgram(online5)
cord.setProgram(online5)

# ------------------------ testing happens here
    ----------------------------------------
```

```
# creates number of shapes
for i in range (9):

        # a single program generation

        # aProgram = newprogram(online1, online5)                #
            pick parents here

        # print(str(aProgram))                                   # see the
            progarm text
        # print("my fitness: ", fitness(aProgram))    # testing
            fitness
        # createImage(str(aProgram), ("code" + str(i)), ("result"
            + str(i)))

        # array for breeding/reproducing
        programarr = [None] * 100

        programreproduce(programarr, online5, online2)
                        # pick parents here
        # currently only doing one generation, can increase
            second param to however many generations wanted
        programbreed(programarr, 0)

        # name result correctly
        createImage(str(programarr[0]), ("code" + str(i)), ("
            result" + str(i)))

        # print the text, fitness
        print(str(programarr[0]))
        print(avgFitness(programarr), "AVG FITNESS")
        print(fitness(programarr[0]), "THIS FITNESS")
```

Bibliography

[Bac96] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[CC13] John Horigan Chris Coyne, Mark Lentczner. context-free. `https://github.com/MtnViewJohn/context-free`, 2013.

[Chr] Chrisc. untitled.

[Chr09] Mikael Hvidtfeldt Christensen. Structural synthesis using a context free design grammar approach. In *Generative Art International Conference*, 2009.

[cra] craftycurate. Ancient map.

[GM13] John S Gero and Mary Lou Maher. *Modeling creativity and knowledge-based creative design*. Psychology Press, 2013.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison−Wesley, 1989.

[Koz92] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[Mit98] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[MN10] Penousal Machado and Henrique Nunes. A step towards the evolution of visual languages. In *First International Conference on Computational Creativity, Lisbon, Portugal*. Citeseer, 2010.

[MNR10] Penousal Machado, Henrique Nunes, and Juan Romero. Graph-based evolution of visual languages. *Applications of Evolutionary Computation*, pages 271–280, 2010.

[mom] momo. Oozescape.

[SP94] Mandavilli Srinivas and Lalit M Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[Whi94] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[zee] zeeaitchison. Nerds tree.